

Deriving approximation tolerance constraints from verification runs^{*}

Tobias Isenberg, Marie-Christine Jakobs, Felix Pauck, and Heike Wehrheim

Paderborn University, Germany
{isenberg,marie.christine.jakobs,wehrheim}@upb.de

Abstract. Approximate computing (AC) is an emerging paradigm for energy-efficient computation. The basic idea of AC is to sacrifice high precision for low energy by allowing for hardware which only carries out “approximately correct” calculations. For software verification, this challenges the validity of verification results for programs run on approximate hardware.

In this paper, we present a novel approach to examine program correctness in the context of approximate computing. In contrast to all existing approaches, we start with a standard program verification and compute the allowed *tolerances* for AC hardware from that verification run. More precisely, we derive a set of constraints which – when met by the AC hardware – guarantees the verification result to carry over to AC. Our approach is based on the framework of *abstract interpretation*. On the practical side, we furthermore (1) show how to extract tolerance constraints from verification runs employing predicate abstraction as an instance of abstract interpretation, and (2) show how to check such constraints on hardware designs. We exemplify our technique on example C programs and a number of recently proposed *approximate adders*.

1 Introduction

Approximate computing (AC) [21,16] is a new computing paradigm which aims at reducing energy consumption at the cost of computation *precision*. A number of application domains can tolerate AC because they are inherently resilient to imprecision (e.g., machine learning, big data analytics, image processing, speech recognition). Computation precision can be reduced by either directly manipulating program executions on the algorithmic level (e.g. by loop perforation [10]) or by employing approximate hardware for program execution [27]. Approximation on the level of hardware can be achieved by techniques like voltage overscaling or by directly making imprecise hardware designs with less chip area. The approximate adders which we will later use employ the latter technique, and simply have limited carry propagation.

For software verification, the use of approximate hardware challenges soundness and raises the question of whether the achieved verification result will really

^{*} This work was partially supported by the German Research Foundation (DFG) within the Collaborative Research Centre “On-The-Fly Computing” (SFB 901).

be valid when the program is being executed. So far, correctness in the context of approximate computing has either studied *quantitative reliability*, i.e., the probability that outputs of functions have correct values [11,24] (employed for the language Rely), or differences between approximate and precise executions [22,14] (applying differential program verification). Alternatively, some approaches plainly use types and type checking to separate the program into precise and approximate parts (language EnerJ) [27]. All of these techniques take a hardware-centric approach: take the (non-)guarantees of the hardware, and develop new analysis methods working under such weak guarantees. The opposite direction, namely use standard program analysis procedures and have the verification impose constraints on the allowed approximation, has not been studied so far. This is despite the fact that such an approach directly allows re-use of existing verification technology for program verification as well as for checking the constraints on the approximate hardware. Another advantage of this approach is that the imposed constraints can be checked on multiple hardware designs, as we did in our examples.

In this paper, we propose a new strategy for making software verification reliable for approximate computing. Within the broad spectrum of AC techniques, we focus on *deterministic* approximate designs, i.e., approximate hardware designs with deterministic truth tables. We start with a verification run proving safety of a program. For the moment on, we assume that the safety property is encoded with assertions or specific error labels `ERR`. With a proper instrumentation of the program various properties can be encoded in such a way. In Section 4 we exemplarily describe how to encode termination proofs.

Our approach derives from that verification run requirements on the hardware executing the program. We call such requirements *tolerance constraints*. A tolerance constraint acts like a pre/postcondition pair and describes the expected output of a hardware design when supplied with specified inputs. The derived tolerance constraints capture the assumptions the verification run has made on the executing hardware. Thus, they are specific to the program and safety property under consideration. Tolerance constraints refer to program statements, e.g., statements using addition as operation. Typically, tolerance constraints are much less restrictive than the precise truth table of a hardware operation would dictate.

To instantiate this general idea, we had to select the underlying verification technique. We discuss the alternatives in Section 7, after we presented our concrete instantiation. In the following, we formulate the derivation of tolerance constraints within the framework of *abstract interpretation*, thus, making the technique applicable to all abstract interpretation based program analyses. We prove soundness of our technique by showing that a program, which has been proven correct, will also run correctly on AC when the employed approximate hardware satisfies the derived tolerance constraints.

To see our technique in practice, we instantiate the general framework based on abstract interpretation with predicate abstraction [15,3]. In this case, tolerance constraints are pairs (p, q) of predicates on inputs and expected outputs

```

int arr[1000];

for(int j:=0;j<990;) {
    j:=j+10;
    if (!(j>=0 && j<1000))
        ERR: ;
    arr[j]:=0;
}

int u:=input();
int sum:=1;

if (u>0)
    sum:=1+u;
if (sum==0)
    ERR: ;

```

Fig. 1. Programs *Array* (left) and *AddOne* (right)

of a hardware operation. As a first example, take a look at the left program in Figure 1. The left program writes to an array within a for-loop. The property to be checked (encoded as an error state `ERR`) is an array-index-inside-bounds check. Using x and y as inputs and z as output (i.e., $z = x + y$), the tolerance constraint on addition (+) derived from a verification run showing correctness is

$$(x \geq 0 \wedge x \leq 989 \wedge y = 10 \Rightarrow z \geq 0 \wedge z \leq 999)$$

It states that the hardware adder should guarantee that adding 10 to a value in between 0 and 989 never brings us outside the range $[0, 999]$, and thus the program never crashes with an index-out-of-bounds exception.

Using the analysis tool CPACHECKER [8] for verification runs, we implemented the extraction of tolerance constraints from abstract reachability graphs constructed during verification. The constraints will be in SMT-Lib format [4]. To complete the picture, we have furthermore implemented a procedure for *tolerance checking* on hardware designs. This technique constructs a specific checker circuit out of a given hardware design (in Verilog) and tolerance constraint. We have evaluated our overall approach on example C programs, e.g., taken from the software verification competition benchmark, using as AC hardware different approximate adders from the literature (Verilog designs taken from the website accompanying [29]). During evaluation, we examined if a program which uses an approximate adder still terminates, adheres to a protocol, or remains memory safe. Additionally, we looked at certain properties of additions like monotonicity to capture the different behavior of precise and approximate adders.

2 Background

We start by formally defining the syntax and semantics of programs, and by introducing the framework of abstract interpretation [13].

Programs. For our formal framework, we assume programs to have integer variables only¹, $Ops = \{+, -, *, \backslash\}$ to be the set of binary operators on integers, \mathbb{Z} to be the integer constants and $Cmp = \{<, \leq, >, \geq, =\}$ the set of comparison operators on integers. Programs use variables out of a set Var , and have two sorts of

¹ For the practical evaluation we, however, allow for arbitrary C programs.

statements from a set Stm : (1) conditionals **assume** b (b boolean condition over Var formed using Ops and Cmp) and (2) assignments $v := \mathbf{expr}$, $v \in Var$, \mathbf{expr} expression over Var formed with Ops . Formally, programs are given by control flow automata.

Definition 1. A control flow automaton (CFA) $P = (L, \ell_0, E, Err)$ consists of a finite set of locations L , an initial location $\ell_0 \in L$ and a set of edges $E \subseteq L \times Stm \times L$ and a set of error locations $Err \subseteq L$.

Note that we mark the error locations in programs with the label **ERR** (or similar). A concrete state of a program is a mapping $s : Var \rightarrow \mathbb{Z}$, and Σ is the set of all states. For a state s , we define a *state update* wrt. $u \in Var$ and $c \in \mathbb{Z}$ to be $s[u := c](u) = c$, $s[u := c](v) = s(v)$ for $u \neq v$. For a state s and a boolean condition b , we write $s \models b$ to state that b is true in s . A configuration of a program is a pair (s, ℓ) , $s \in \Sigma$, $\ell \in L$.

The semantics of program statements is given by the following (partial) *next transformers* $next_{stm} : \Sigma \rightarrow \Sigma$ with

$$next_{stm}(s) = s' \text{ with } \begin{cases} s' = s & \text{if } stm \equiv \mathbf{assume } b \wedge s \models b \\ s' = s[v := s(\mathbf{expr})] & \text{if } stm \equiv v := \mathbf{expr} \end{cases}$$

We lift $next_{stm}$ to sets of states by $next_{stm}(S) = \{next_{stm}(s) \mid s \in S\}$. Note that this lifted function is total. The next transformers together with the control flow determine the transition system of a program.

Definition 2. The concrete transition system $T(P) = (Q, q_0, \rightarrow)$ of a CFA $P = (L, \ell_0, E, Err)$ consists of

- a set of configurations $Q = \Sigma \times L$,
- an initial configuration $q_0 = (s_0, \ell_0)$ where $s_0(v) = 0$ for all $v \in Var$,
- a transition relation $\rightarrow \subseteq Q \times Stm \times Q$ with $(s, \ell) \xrightarrow{stm} (s', \ell')$ if $(\ell, stm, \ell') \in E$ and $next_{stm}(s) = s'$.

An error location is *reachable* in $T(P)$ if there is a path from (s_0, ℓ_0) to a configuration $(*, \ell)$ with $\ell \in Err$. If no error location is reachable, we say that the transition system is *free of errors*.

Abstract interpretation. For verifying that a program is free of errors, we use the framework of abstract interpretation (AI) [13]. Thus we assume that the verification run from which we derive tolerance constraints is carried out by an analysis tool employing abstract interpretation as basic verification technology.

Instead of concrete states, instances of AI frameworks employ abstract domains Abs and execute abstract versions of the next transformers on it. Abstract domains are equipped with an ordering \sqsubseteq_{Abs} , and (Abs, \sqsubseteq_{Abs}) has to form a complete lattice (as does $(2^\Sigma, \subseteq)$). To relate abstract and concrete domain, two monotonic functions are used: an abstraction function $\alpha : 2^\Sigma \rightarrow Abs$ and a concretisation function $\gamma : Abs \rightarrow 2^\Sigma$. The pair (α, γ) has to form a *Galois connection*, i.e. the following has to hold: $\forall S \in 2^\Sigma : S \subseteq \gamma(\alpha(S))$ and

$\forall abs \in Abs : \alpha(\gamma(abs)) \sqsubseteq_{Abs} abs$. We require the least element of the lattice (Abs, \sqsubseteq_{Abs}) (which we denote by a_\perp) to be mapped onto the least element of $(2^\Sigma, \subseteq)$ which is the empty set \emptyset .

On the abstract domain, the AI instance defines a total abstract next transformer $next_{stm}^\# : Abs \rightarrow Abs$. To be useful for verification, the abstract transformer has to faithfully reflect the behaviour of the concrete transformer.

Definition 3. *An abstract next transformer $next_{stm}^\# : Abs \rightarrow Abs$ is a safe approximation of the concrete next transformer if the following holds:*

$$\forall abs \in Abs, \forall stm \in Stm : \alpha(next_{stm}(\gamma(abs))) \sqsubseteq_{Abs} next_{stm}^\#(abs)$$

Using the abstract next transformer, we can construct an abstract transition system of a program.

Definition 4. *The abstract transition system $T^\#(P) = (Q, q_0, \rightarrow)$ of a program P with respect to an abstract domain Abs and functions $next_{stm}^\#$ consists of*

- a set of configurations $Q = Abs \times L$,
- an initial configuration $q_0 = (a_0, \ell_0)$ where $a_0 = \alpha(\{s_0\})$ with $s_0(v) = 0$ for all $v \in Var$,
- a transition relation $\rightarrow \subseteq Q \times Stm \times Q$ with $(a, \ell) \xrightarrow{stm} (a', \ell')$ if $(\ell, stm, \ell') \in E$ and $next_{stm}^\#(a) = a'$.

An abstract configuration (a, ℓ) is *reachable* in $T^\#(P)$ if there is a path from $q_0 = (a_0, \ell_0)$ to a configuration (a, ℓ) . We denote the set of reachable configurations in $T^\#(P)$ by $Reach(T^\#(P))$ or simply $Reach^\#$. An error location is reachable in $T^\#(P)$ if there is a path from (a_0, ℓ_0) to a configuration (a, ℓ) with $\ell \in Err$, $a \neq a_\perp$. Note that we allow paths to configurations (a_\perp, ℓ) , $\ell \in Err$, since a_\perp represents the empty set of concrete states, and thus does not stand for a concretely reachable error.

The abstract transition system can be used for checking properties of the concrete program whenever the abstract transformers are safe approximations.

Theorem 1. *Let P be a CFA, T its concrete and $T^\#$ its abstract transition system according to some abstract domain and safe abstract next transformer. Then the following holds: If $T^\#$ is free of errors, so is T .*

3 Transformer Constraints

The framework of abstract interpretation is used to verify that a program is free of errors. To this end, the abstract transition system is build and inspected for reachable error locations. However, the construction of the abstract transition system and thus the soundness of verification relies on the fact that the abstract transformer safely approximates the concrete transformer, and this in particular means that we verify properties of a program execution using the concrete

transformers for next state computation. This assumption is not true anymore when we run our programs on approximate hardware.

For a setting with approximate hardware, we have *approximate next transformers* $next_{stm}^{AC}$ for (some or all) of our statements. The key question is now the following: Under which conditions on these approximate transformers will our verification result carry over to the AC setting? To this end, we need to find out what "properties" of a statement the verification run has actually used. This can be seen in the abstract transition system by looking at the transitions labelled with a specific statement, and extracting the abstract states before and after this statement. A *tolerance constraint* for a statement includes all such pairs of abstract states, specifying a number of pre- and postconditions for the statement.

Definition 5. Let $T^\# = (Q, q_0, \rightarrow)$ be an abstract transition system of a program P , $stm \in Stm$ a statement. Let $((a_1^i, \ell_1^i) \xrightarrow{stm} (a_2^i, \ell_2^i))_{i \in I}$ be the family of transitions in $\xrightarrow{stm} \cap (Reach^\# \times Reach^\#)$.

The tolerance constraint for stm in $T^\#$ is the family of pairs of abstract states $((a_1^i, a_2^i))_{i \in I}$.

While the concrete transformers by safe approximation fulfill all these constraints, the approximate transformers might or might not adhere to the constraints.

Definition 6. A next transformer $next_{stm}^{AC} : \Sigma \rightarrow \Sigma$ fulfills a tolerance constraint $((a_1^i, a_2^i))_{i \in I}$ if the following property holds for all $i \in I$:

$$s \in \gamma(a_1^i) \Rightarrow next_{stm}^{AC}(s) \in \gamma(a_2^i) .$$

When programs are run on approximate hardware, the execution will use some approximate and some precise next transformers depending on the actual hardware. For instance, the execution might employ an approximate adder, and thus all statements using addition will be approximate. We let $T^{AC}(P)$ be the transition system of program P constructed by using $next_{stm}^{AC}$ for the approximate statements and standard concrete transformers for the rest. This lets us now formulate our main theorem about the validity of verification results on AC hardware.

Theorem 2. Let P be a program and $next_{stm}^{AC}$ be a next transformer for stm fulfilling the tolerance constraint on stm derived from an abstract transition system $T^\#(P)$ wrt. some abstract domain Abs and safe abstract next transformers. Then we get:

If $T^\#(P)$ is free of errors, so is $T^{AC}(P)$.

Proof: Let $((a_1^i, a_2^i))_{i \in I}$ be the tolerance constraint for stm in $T^\#(P)$. Assume the contrary, i.e., there is a path to an error location in T^{AC} : $(s_0, \ell_0) \xrightarrow{stm_0} (s_1, \ell_1) \xrightarrow{stm_1} \dots \xrightarrow{stm_{n-1}} (s_n, \ell_n)$ such that $\ell_n \in Err$. We show by induction that there exists a path $(a_0, \ell_0) \xrightarrow{stm_0} (a_1, \ell_1) \xrightarrow{stm_1} \dots \xrightarrow{stm_{n-1}} (a_n, \ell_n)$ in the abstract transition system $T^\#$ such that $s_j \in \gamma(a_j)$.

Induction base. $s_0 \in \gamma(a_0)$ since $a_0 = \alpha(\{s_0\})$ and $\{s_0\} \subseteq \gamma(\alpha(\{s_0\}))$ by Galois connection properties.

Induction step. Let $s_j \in \gamma(a_j)$, $next_{stm_j}(s_j) = s_{j+1}$ and $(\ell_j, stm_j, \ell_{j+1}) \in E$.

Let $S_j = \gamma(a_j)$ (hence $s_j \in S_j$). Now we need to consider two cases:

Case (1): $stm_j \neq stm$: Then the next transformer applied to reach the next configuration is the standard transformer. Thus let $next_{stm_j}(\gamma(a_j)) = S_{j+1}$.

By safe approximation of $next^\#$, we get $\alpha(S_{j+1}) \sqsubseteq_{Abs} next^\#_{stm_j}(a_j) = a_{j+1}$. By monotonicity of γ : $\gamma(\alpha(S_{j+1})) \subseteq \gamma(a_{j+1})$. By Galois connection: $S_{j+1} \subseteq \gamma(\alpha(S_{j+1}))$. Hence, by transitivity $s_{j+1} \in \gamma(a_{j+1})$.

Case (2): $stm_j = stm$: Let $next^\#_{stm}(a_j) = a_{j+1}$. By definition of tolerance constraint extraction, the pair (a_j, a_{j+1}) has to be in the family of tolerance constraints, i.e., $\exists i \in I : (a_j, a_{j+1}) = (a_1^i, a_2^i)$. Since $next_{stm}^{AC}$ fulfills the constraint, $next_{stm_j}^{AC}(s_j) \in \gamma(a_{j+1})$. \square

4 Preserving termination

So far, we have been interested in the preservation of already proven safety properties on approximate hardware. Another important issue is the preservation of *termination*: whenever we have managed to show that a program terminates on precise hardware, we would also like to get constraints that guarantee termination on AC hardware. In order to extend our approach to termination, we make use of an approach for encoding termination proofs as safety properties [12].

We start with explaining standard termination proofs. Nontermination arises when we have loops in programs and the loop condition never gets false in a program execution. In control flow automata, a *loop* is a sequence of locations ℓ_0, \dots, ℓ_n such that there are statements stm_i , $i = 0..n-1$, with $\ell_i \xrightarrow{stm_i} \ell_{i+1}$ and $\ell_0 = \ell_n$. In this, a location $\ell = \ell_i$ is said to be on the loop. Every well-structured loop has a condition and a loop body: the *start* of a loop body is a location ℓ such that there are locations ℓ', ℓ'' and a boolean condition b s.t. $\ell' \xrightarrow{assume\ b} \ell$ and $\ell' \xrightarrow{assume\ !b} \ell''$ are in the CFA and ℓ' is on a loop, but either ℓ'' is not on a loop, or is on a different loop. Basically, we just consider CFAs of programs constructed with while or for constructs, not with gotos or recursion. However, the latter is also possible when the verification technique used for proving termination covers such programs.

Definition 7. A non-terminating run of a CFA $P = (L, \ell_0, E, Err)$ is an infinite sequence of configurations and statements $(s_0, \ell_0) \xrightarrow{stm_1} (s_1, \ell_1) \xrightarrow{stm_2} \dots$ in the transition system $T(P)$. If P has no non-terminating runs, then P terminates.

Proposition 1. In every non-terminating run, at least one loop start ℓ occurs infinitely often.

We assume some standard technique to be employed for proving termination. Such techniques typically consist of (a) the synthesis of a termination argument, and (b) the check of validity of this termination argument. Termination

arguments are either given as monolithic ranking functions or as disjunctively well-founded transition invariants [26]. Here, we will describe the technique for monolithic ranking functions.

1. For every loop starting in ℓ , define a *ranking function* f_ℓ on the program variables, i.e., $f_\ell : \Sigma \rightarrow W$, where (W, \leq) is a well-founded order with least element \perp_W .
2. Show f_ℓ to decrease with every loop execution, i.e., if $(s, \ell) \xrightarrow{stm_1} (s_1, \ell_1) \xrightarrow{stm_2} \dots \xrightarrow{stm_n} (s', \ell)$ is a path in $T(P)$, show $f_\ell(s') < f_\ell(s)$.
3. Show f_ℓ to be greater or equal than the least element of W at loop start, i.e., for all starts of loop bodies ℓ and $(s, \ell) \in Q_{T(P)}$, show $f_\ell(s) \geq \perp_W$.

If properties (2) and (3) hold, we say that the ranking function is *valid*. Note that we are not interested in computing ranking functions here; we just want to make use of existing verification techniques. The following proposition states a standard result for ranking functions (see e.g. [23,20]).

Proposition 2. *Let P be a program. If every loop ℓ of P has a valid ranking function, then P terminates.*

As an example consider the program *Sum* on the left of Figure 2. It computes the sum of all numbers from 0 up to some constant N . It terminates since variable i is constantly increased. As ranking function we can take $N - i$ using the well-founded ordering \mathbb{N} .

In order to encode the above technique in terms of assertions, we instrument a program P along the lines used in the tool TERMINATOR [12] thereby getting a program \hat{P} as follows. Let $Var = \{x_1, \dots, x_n\}$ be the set of variables occurring in the program. At starts of loop bodies ℓ we insert

```
if (!(f_l(x1, ..., xn) >= bot_W)
  ERR:
old_x1 := x1;
...
old_xn := xn;
```

and at loop ends we insert

```
if (!(f_l(x1, ..., xn) <_W f_l(old_x1, ..., old_xn))
  ERR:
```

when given a ranking function f_ℓ and a well-founded ordering $(W, <_W)$ with bottom element bot_W .

Proposition 3. *If \hat{P} is free of errors, then P terminates.*

Hence we can use standard safety proving for termination as well (once we have a ranking function), and thereby derive tolerance constraints. In the left of Figure 2 we see the instrumented version of program *Sum*. Here, we have already applied an optimization: we only make a copy of variable i since the ranking function only refers to i and N , and N does not change anyway.

<pre> sum=0; i=0; while (i<N) { sum=sum+i; i=i+1; } </pre>	<pre> sum=0; i=0; while (i<N) { if (!(N-i > 0)) ERR: ; old_i=i; sum=sum+i; i=i+1; if (!(N-i < N-old_i)) ERR: ; } </pre>
---	--

Fig. 2. Program *Sum* (on the left) and its instrumented version (on the right).

5 Constraint Extraction for Predicate Analysis

Section 3 has formally defined the extraction of tolerance constraints from abstract transition systems and has proven its soundness. Now, we will take a closer look at constraint extraction in practice. To this end, we choose an instance of the abstract interpretation framework, namely predicate abstraction [15,3]. Furthermore, instead of deriving constraints for statements, we derive constraints for *operators* since in practice we do not have specific hardware for whole statements but just for the operations used in expressions within a statement.

We start with defining predicate abstraction. For this, we fix a set of predicates \mathcal{P} over Var, Cmp, \mathbb{Z} and Ops . In practice, these predicates will be incrementally computed by a counter-example-guided abstraction refinement approach [17] which we just assume to exist (and which is provided by the tool that we employ for our experiments). We define $\neg\mathcal{P} := \{\neg p \mid p \in \mathcal{P}\}$ and let the abstract domain *Abs* be conjunctions of predicates or their negations (also directly written as set of literals, hence \emptyset is true, $\mathcal{P} \cup \neg\mathcal{P}$ is false):

$$(\{\bigwedge_{q \in Q} q \mid Q \subseteq \mathcal{P} \cup \neg\mathcal{P}\}, \Rightarrow)$$

The Galois connection is given by letting $\alpha(S) = \{q \in \mathcal{P} \cup \neg\mathcal{P} \mid \forall s \in S : s \models q\}$ and $\gamma(Q) := \{s \in \Sigma \mid \forall q \in Q : s \models q\}$. We write $s \models Q$ iff $s \models q$ for all $q \in Q$. For the definition of the abstract next transformers see for instance [3]. Note that tolerance constraints in this domain take the form $(Q_1, Q_2), Q_j \subseteq \mathcal{P} \cup \neg\mathcal{P}, j \in \{1, 2\}$.

This abstract domain can be used to show program *Array* from Figure 1 to be free of errors. Figure 3 shows the abstract transition system of program *Array* using the predicate set $\mathcal{P} = \{j \geq 0, j \leq 989, j \leq 999\}$. The predicates holding in an abstract configuration (a, ℓ) , i.e., the abstract state a , are written next to the purple location. We see that the location labeled *err* occurs in the graph, but the abstract state in this configuration is $a_{\perp} = false$, and, thus, we say that this error is not reachable.

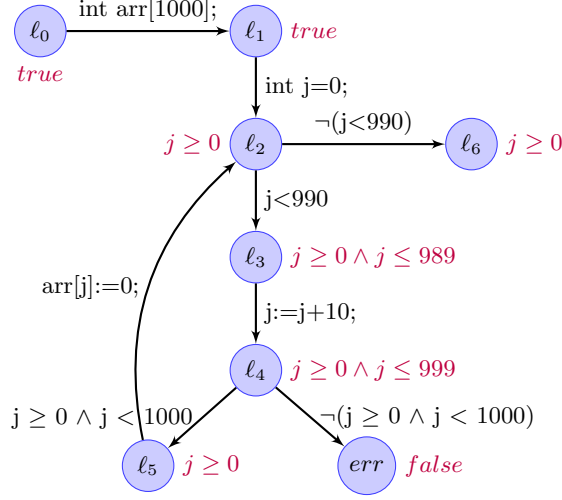


Fig. 3. Abstract transition system of program *Array*

For the extraction of tolerance constraints for operators $op \in Op$, we assume our statements to take the form of *three-address code* (3AC) [1]. In three-address code form, all operators op occur in programs only in statements $v := a \text{ op } b$, where a and b are variables or constants. Every program can be brought in such a form (e.g., intermediate representations generated during compilation take this form). We use this 3AC form because we need to isolate operators, and only have statements with one (possibly approximate) operator in. Note that program *Array* is in 3AC form.

Furthermore, the tolerance constraints, i.e., pre- and postcondition predicates, derived from abstract transition systems are specified over the *program variables*. As an example, take the operator $+$. In the program *Array* this operator occurs in the statement $j := j + 10$. The tolerance constraint for this statement derived from the abstract transition system in Figure 3 is $(j \geq 0 \wedge j \leq 989, j \geq 0 \wedge j \leq 999)$. This constraint refers to the program variable j . If the approximate adder used for $+$ has inputs x and y and output z , this constraint first of all needs to be brought into a form using variables x , y and z . This is achieved using the following replacement operator.

Definition 8. Let $Q \in \mathcal{P} \cup \neg\mathcal{P}, p \in Q, v_1, v_2 \in Var$. The predicate $p[v_2 \triangleright v_1]$ is obtained from p by replacing all occurrences of v_2 by v_1 . We lift this to sets by letting $Q[v_2 \triangleright v_1] := \{q[v_2 \triangleright v_1] \mid q \in Q\}$. For constants $c \in \mathbb{Z}$, we define $Q[c \triangleright v_1] := Q \cup \{v_1 = c\}$.

Proposition 4. For all $q \in \mathcal{P} \cup \neg\mathcal{P}$ such that $x \notin \text{vars}(q)$:

$$s[x := s(u)] \models q[u \triangleright x] \quad \Leftrightarrow \quad s \models q$$

For constraint $(Q_1, Q_2) = (j \geq 0 \wedge j \leq 989, j \geq 0 \wedge j \leq 999)$, statement $j := j + 10$ and adder with inputs x and y , output z , the replacement we need to make is $(Q_1[j \triangleright x, 10 \triangleright y], Q_2[j \triangleright z]) = (x \geq 0 \wedge x \leq 989 \wedge y = 10, z \geq 0 \wedge z \leq 999)$. This is the constraint which ultimately needs to be checked for the approximate hardware. In the following we assume all binary operators to have signature $(x : \mathbb{Z}, y : \mathbb{Z}) \rightarrow (z : \mathbb{Z})$, x, y and z to not occur as variables in the program nor in the predicates and use (\hat{Q}_1, \hat{Q}_2) to refer to the constraints obtained after the replacement.

Definition 9. An approximate operator $op^{AC} : \mathbb{Z} \times \mathbb{Z} \rightarrow \mathbb{Z}$ adheres to a tolerance constraint (\hat{Q}_1, \hat{Q}_2) (i.e., over x, y and z) if

$$\forall s \in \Sigma : s \models \hat{Q}_1 \Rightarrow s[z := op^{AC}(s(x), s(y))] \models \hat{Q}_2$$

Adherence to constraints by operators implies adherence to constraints by statements using these operators.

Lemma 1. Let (Q_1, Q_2) be a tolerance constraint extracted from $T^\#$ for $stm \equiv u := v \text{ op } w$. If op^{AC} adheres to $(Q_1[v \triangleright x, w \triangleright y], Q_2[u \triangleright z])$, then $next_{stm}^{AC} \equiv u := v \text{ op } w$ adheres to (Q_1, Q_2) .

Proof: We need to show that $next_{stm}^{AC}$ adheres to (Q_1, Q_2) . We first of all take the definition of it and rewrite it a little.

$$\begin{aligned} s \in \gamma(Q_1) &\Rightarrow next_{stm}^{AC}(s) \in \gamma(Q_2) \\ &\Leftrightarrow \{ \text{definition of } next_{stm}^{AC} \} \\ s \in \gamma(Q_1) &\Rightarrow s[u := op^{AC}(s(v), s(w))] \in \gamma(Q_2) \\ &\Leftrightarrow \{ \text{definition of } \gamma \} \\ s \models Q_1 &\Rightarrow s[u := op^{AC}(s(v), s(w))] \models Q_2 \end{aligned}$$

The last implication is now shown as follows:

$$\begin{aligned} s \models Q_1 &\Rightarrow s[x := s(v), y := s(w)] \models Q_1[v \triangleright x, w \triangleright y] \\ &\Rightarrow s[x := s(v), y := s(w), z := op^{AC}(s(v), s(w))] \models Q_2[u \triangleright z] \\ &\Rightarrow s[x := s(v), y := s(w), u := op^{AC}(s(v), s(w))] \models Q_2 \\ &\Rightarrow s[u := op^{AC}(s(v), s(w))] \models Q_2 \end{aligned}$$

□

This finally gives us our main soundness result for predicate analysis which is an immediately corollary of Lemma 1 and Theorem 2.

Corollary 1. Let $T^\#$ be an abstract transition system constructed using safe approximations and let all approximate operators op^{AC} adhere to the constraints derived from $T^\#$. Then: If $T^\#$ is free of errors, so is T^{AC} .

Implementation. As proof of concept we integrated our proposed constraint extraction into the software analysis tool CPACHECKER [8], a tool for C program analysis which is configurable to abstract interpretation based analyses. Mainly, we added a constraint extraction algorithm plus some additional helper classes. Our constraint extraction algorithm builds on top of CPACHECKER’s predicate analysis which uses the technique of adjustable block encoding [7], a technique which allows to specify at which locations an abstraction should be computed. For our extraction we need to make sure that we have an abstract state immediately before and after each statement which uses the operation of interest op^2 . To identify these abstraction points and later the tolerance constraints, we first need to identify the statements using the operation op . Afterwards, we run CPACHECKER’s standard predicate analysis which provides us with an abstract reachability graph (ARG), a structure similar to the abstract transition system. In the ARG, the predicates are given in SMT-Lib format [4] since CPACHECKER is using state-of-art SMT solvers for predicate analysis. From the ARG, we extract the tolerance constraints and write one SMT file per constraint (Q_1, Q_2) which is in the input format required by our next tool building the hardware checker. The SMT file mainly contains the description of (Q_1, Q_2) pairs plus additional information about the signature of the statement for which the constraint was extracted. The signature is needed by the next tool to construct $(\widehat{Q}_1, \widehat{Q}_2)$.

To run the tolerance constraint extraction within CPACHECKER, one can use the configuration file `predicateAnalysis-ToleranceConstraintsExtraction-PLUS.properties` that we used in our evaluation to extract tolerance constraints for additions.

6 Constraint Checking

The final step of our technique is the check of the extracted constraints on actual hardware designs of approximate operations. For simplicity of representation, we restrict the following explanations to the case of a single constraint³. The input to the checking phase thus consists of a constraint (Q_1, Q_2) , an approximate operator op^{AC} and the corresponding program statement $u := v \text{ op } w$. The checking of the constraint on a given hardware design with inputs x, y and output z (in our case specified in Verilog) proceeds in three steps:

Mapping The mapped tolerance constraint $(\widehat{Q}_1, \widehat{Q}_2) = (Q_1[v \triangleright x, w \triangleright y], Q_2[u \triangleright z])$ is constructed. As a result, the tolerance constraint $(\widehat{Q}_1, \widehat{Q}_2)$ uses the variables x, y and z when referring to the inputs and output of op^{AC} . Additional variables of the program (besides u, v and w) may still occur in the constraint which are not used in the hardware design. We denote these variables as *side variables*.

² The operation of interest is made configurable in CPACHECKER.

³ A generalization to a family of constraints is straightforward.

Transformation The mapped constraint is transformed into Verilog code giving a *checker circuit*. The checker circuit is created as Verilog code in two steps. First, the logical formulae of the tolerance constraints are compiled to Verilog code (see [25]). In this, side variables are treated like other inputs. We then fix a single output of the checker called *error* by setting $error := \neg(\hat{Q}_1 \Rightarrow \hat{Q}_2)$.

Combination The generated tolerance constraint checker is afterwards combined with the hardware design of op^{AC} into an *adherence checker*. For our examples, the AC hardware designs are also given in Verilog. The combination is done using a top module that contains and wires the design of op^{AC} and the tolerance checker as sub-modules. The wiring is done as depicted in Figure 4.

The resulting circuit is afterwards checked for safety, i.e., that for no combinations of values on the primary inputs the error flag is raised. This step can be done using standard hardware verification techniques (unsatisfiability checking).

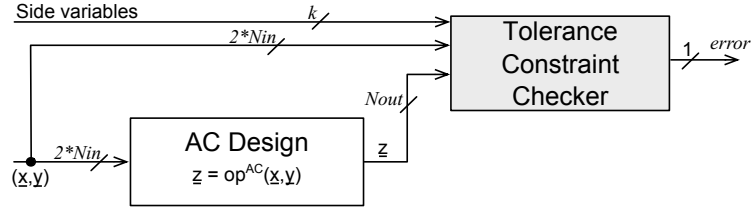


Fig. 4. Adherence Checker combining AC design with Tolerance Constraint Checker

As an example, consider again program *Array* given on the left side of Figure 1. The tolerance constraint extracted for operator $+$ is $(j \geq 0 \wedge j \leq 989, j \geq 0 \wedge j \leq 999)$ and the program statement is $j := j + 10$. In SMT-Lib format, the constraint is

```

(define-fun Q_1 () Bool (and (<= 0 |main::j|) (<= |main::j| 989)))
(define-fun Q_2 () Bool (and (<= 0 |main::j@1|)
                             (<= |main::j@1| 999)))

```

The structural mapping of the variables is represented as $[|main :: j| \triangleright x]$, $[10 \triangleright y]$ and $[|main :: j@1| \triangleright z]$. As a result, the mapped constraint can be represented as follows.

```

(define-fun mappedQ_1 () Bool (and (and (<= 0 x) (<= x 989))
                                   (= y 10)))
(define-fun mappedQ_2 () Bool (and (<= 0 z) (<= z 999)))

```

Figure 5 gives the Verilog code of the checker circuit belonging to this mapped constraint. Note that the length of the input vectors have to be adapted to fit the one provided by the hardware design of op^{AC} .

```

module TCChecker(x,y,z, error);
    parameter Nin = 32;
    parameter Nout = 33;

    input [Nin-1:0] x;
    input [Nin-1:0] y;
    input [Nout-1:0] z;
    output error;

    wire term__1;
    wire term__2;
    wire term__3;
    wire term__4;
    wire Q__1;
    wire Q__2;
    wire pre__gen_0;

    assign term__1 = (x <= 989);
    assign term__2 = (0 <= x);
    assign Q__1 = (term__2 && term__1);
    assign term__3 = (z <= 999);
    assign term__4 = (0 <= z);
    assign Q__2 = (term__4 && term__3);
    assign pre__gen_0 = (y == 10);

    assign error = !((!(Q__1 && pre__gen_0) || Q__2));
endmodule

module AdherenceChecker(
    input [31:0] inp1,
    input [31:0] inp2,
    output errorBit);

    wire[32:0] outp;

    Adder add(
        .x(inp1),
        .y(inp2),
        .z(outp)
    );

    TCChecker check(
        .x(inp1),
        .x(inp2),
        .z(outp),
        .error(errorBit)
    );
endmodule

```

Fig. 5. Verilog code of checker circuit (left) and its combination with Adder (right)

Experiments. In our experiments, we used the software analysis tool CPACHECKER to extract the tolerance constraints from a verification run. We employed the tools *Yosys* [31] and *ABC* [5] for synthesis and generation of a CNF formula that encodes the value of the error flag in dependence on all the inputs. Using PicoSAT [9], we checked the unsatisfiability of the formula, denoting that the error flag is never raised, i.e., the tolerance constraints are met by the implementation.

In the following, we give the results of our experiments. In our experiments we studied tolerance constraints for addition (since this is the only operation for which approximate hardware is currently publicly available). While it is often accepted that in approximate computing a computation result is not functional equivalent with a precise computation result, a approximate computation must still well-behave. For example, memory accesses should remain safe or it should still terminate or stick to a certain protocol. That is why, during program verification we considered one of these properties instead of functional behavior.

We extracted tolerance constraints from the verification of a number of hand-crafted programs (including our three examples) and some programs from the

subcategory **ControlFlow** and **ProductLines** of the SV-COMP⁴ [6]. We chose our programs to get tolerance constraints from a variety of verification problems and are very well aware that these programs are no typical candidates for approximate computing. The handcrafted programs **AddOne**, **EvenSum**, and **MonotonicAdd** should examine the addition of positive numbers. Programs **sum**, **quotient**, and **mirror_matrix** use the previously described technique to encode termination proofs with assertions. To artificially enforce a difference between the behavior of the approximate adder, we used program **SpecificAdd** which checks that the addition of $30 + 50$ is indeed 80. The programs from the SV-COMP (the last 10 programs shown in Table 1) check protocol properties, e.g. correct locking behavior.

We checked the tolerance constraints on a standard, non-approximate ripple carry adder (RCA) and a set of approximate adders provided by the Karlsruhe library of [29] (called ACA-I [30], ACA-II (ACA_II_N16_Q4)[19], ETAI [33], GDA [32] and GeAr). Table 1 shows our results. For each program, we show the number of additions $\#+$, the number of program statements $\#stm$, the number of constraints extracted $\#tc$ and whether an adder meets the tolerance constraints \checkmark or not \times .

Table 1. Results of experiments

program	$\#+$	$\#stm$	$\#tc$	RCA	ACA-I	ACA-II	ETAI	GDA	GeAr
Array	1	15	1	\checkmark	\checkmark	\checkmark	\checkmark	\checkmark	\checkmark
AddOne	1	22	1	\checkmark	\times	\times	\times	\times	\times
Attach/Detach	1	26	1	\checkmark	\checkmark	\checkmark	\checkmark	\checkmark	\checkmark
EvenSum	2	24	4	\checkmark	\times	\times	\times	\times	\times
MonotonicAdd	1	20	1	\checkmark	\times	\times	\times	\times	\times
SpecificAdd	1	13	1	\checkmark	\checkmark	\times	\checkmark	\checkmark	\checkmark
sum	2	26	2	\checkmark	\times	\times	\times	\times	\times
quotient	2	35	2	\checkmark	\times	\times	\times	\times	\times
mirror_matrix	2	42	2	\checkmark	\times	\times	\times	\times	\times
locks_5	5	114	31	\checkmark	\checkmark	\checkmark	\checkmark	\checkmark	\checkmark
locks_8	8	171	255	\checkmark	\checkmark	\checkmark	\checkmark	\checkmark	\checkmark
cdaudio	13	1888	23	\checkmark	\checkmark	\checkmark	\checkmark	\checkmark	\checkmark
diskperf	19	981	12	\checkmark	\checkmark	\checkmark	\checkmark	\checkmark	\checkmark
floppy4	31	1370	34	\checkmark	\checkmark	\checkmark	\checkmark	\checkmark	\checkmark
kbfiltr2	11	759	15	\checkmark	\checkmark	\checkmark	\checkmark	\checkmark	\checkmark
minepump_s5_p64	2	741	3	\checkmark	\checkmark	\checkmark	\checkmark	\checkmark	\checkmark
minepump_s5_simulator	2	811	3	\checkmark	\checkmark	\checkmark	\checkmark	\checkmark	\checkmark
clnt_4	13	575	18	\checkmark	\checkmark	\checkmark	\checkmark	\checkmark	\checkmark
svr_8	19	668	14	\checkmark	\checkmark	\checkmark	\checkmark	\checkmark	\checkmark

⁴ Some additions first had to be brought in three-address code form and in some programs we replaced some constant assignments by proper addition.

Our first observation is that except for program `SpecificAdd` which we created to show a different behavior between the approximate adders either all approximate adders meet the extracted tolerance constraint or none of them. This is because all approximate adders use the same principle: reduction of the carry chain. In their addition, they use a set of subadders and the carry bit of the previous subadder is either dropped or imprecisely predicted. The effect of this reduction only shows off for specific numbers and these specific numbers differ among the approximate adders. Hence, adding 30 and 50 failed only in the approximate adders ACA-II.

Interestingly, the approximate adders meet the extracted tolerance constraints for all of the SV-COMP programs. On the one hand, not all additions in the programs have an effect on the correctness of the program (and thus verification imposes no constraints on them). On the other hand, typically the additions considered during verification which had an effect increase a variable value in the range $[0, 9]$ by one which can be computed precisely by the first subadder of all approximate adders.

For our own programs, one can see that all sorts of cases occur: all approximate adders satisfy the extracted constraints (as is the case for program `Array`), some do and some do not (on program `SpecificAdd`), and all do not. An instance of the latter case is our example program `AddOne` from the right of Figure 1. The variable u which is increased by 1 can be any positive integer (it is an input). The derived constraint for operator $+$ is $((1 \leq u)[u \triangleright x, 1 \triangleright y], (sum \neq 0)[sum \triangleright z]) = (1 \leq x \wedge y = 1, z \neq 0)$. For our verification of the property, we require that the increase of that variable does not result in value zero, which can be the case if the carry propagation is imprecise. Thus, here the approximate designs fail to satisfy the constraint. Hence, an execution of the program on approximate hardware with these adders could reach the error state. The imprecise carry propagation is also the reason why the approximate adders cannot guarantee termination of programs `sum`, `quotient`, and `mirror_matrix`. For termination all three programs rely on an addition which is strongly monotonic up to a certain threshold (maximal int value). However, due to the imprecise carry propagation an addition of two positive integers may result in value zero.

7 Discussion

To compute requirements on AC hardware with the help of program verification, further approaches are conceivable. For example, one could model the approximate operation as a function call. This means, the approximate operations in a program, e.g. the approximate addition, must be replaced by a call to corresponding function. Now, one applies a verification technique, e.g. [28], which computes function summaries [18]. The function summary for the approximate operation, in principle a description of a pre-/postcondition pair, gives us the constraint on the AC hardware.

In another alternative one would also model the approximate operation as a function call, but now one assumes that the behavior of the function approx

modeling the approximate operation is unknown. In this case, one may use a technique like [2] which tries to generate the weakest specification for the function approx which still ensures program correctness w.r.t. the desired property. The specification for the function approx which is in principle an encoding of a pre-/postcondition pair describes the requirement on the AC hardware.

We are confident that both alternatives could be used with our general approximation tolerance constraints approach. To use those alternatives the function summary and the inferred specification must be transformed into a tolerance constraint checker. We think this is feasible because [28] and [2] already seem to use logic formulae to express the function summary and the specification.

For this paper, we decided to use abstract interpretation as a first example to generate the tolerance constraints for AC hardware. The disadvantage of abstract interpretation is that we might get multiple constraints. The two alternatives only generate one constraint. In practice, we solved this problem such that multiple constraints are conjuncted into a single constraint during the generation of the tolerance constraint checker. On the other hand, we do not need to transform the approximate operations into function calls and the generation of three-address code is rather standard for compilers. Another reason is that we are already familiar with abstract interpretation. Additionally, the verification tool CPACHECKER which we typically use for verification is based on abstract interpretation and analyses functions via in-lining.

8 Conclusion

In this paper, we have proposed a new way of making software verification robust against approximate hardware. Its basic principle is the derivation of constraints on AC hardware from verification runs. We have shown our technique to be sound, i.e., shown that the verification result carries over to a setting with AC hardware when the hardware satisfies the derived constraints. First experimental results have shown that the verification result often but not always carries over. More experiments are, however, necessary when further AC implementations of operations – besides approximate adders – become available.

References

1. Aho, A.V., Sethi, R., Ullman, J.D.: *Compilers: Principles, Techniques, and Tools*. Addison-Wesley (1986)
2. Albarghouthi, A., Dillig, I., Gurfinkel, A.: Maximal specification synthesis. In: *POPL*. pp. 789–801. ACM (2016)
3. Ball, T., Podelski, A., Rajamani, S.K.: Boolean and cartesian abstraction for model checking C programs. *STTT* 5(1), 49–58 (2003)
4. Barrett, C., Fontaine, P., Tinelli, C.: *The SMT-LIB Standard: Version 2.5*. Tech. rep., Department of Computer Science, The University of Iowa (2015), available at <http://www.SMT-LIB.org>
5. Berkeley, ABC: A system for sequential synthesis and verification (2005)

6. Beyer, D.: Software verification and verifiable witnesses. In: Baier, C., Tinelli, C. (eds.) TACAS, LNCS, vol. 9035, pp. 401–416. Springer Berlin Heidelberg (2015)
7. Beyer, D., Keremoglu, M.E., Wendler, P.: Predicate abstraction with adjustable-block encoding. In: FMCAD. pp. 189–198. FMCAD Inc (2010)
8. Beyer, D., Keremoglu, M.: CPAchecker: A Tool for Configurable Software Verification. In: CAV, pp. 184–190. LNCS, Springer (2011)
9. Biere, A.: Picosat. <http://fmv.jku.at/picosat> (2013)
10. Carbin, M., Kim, D., Misailovic, S., Rinard, M.C.: Verified integrity properties for safe approximate program transformations. In: Albert, E., Mu, S. (eds.) Workshop on Partial Evaluation and Program Manipulation. pp. 63–66. ACM (2013)
11. Carbin, M., Misailovic, S., Rinard, M.C.: Verifying quantitative reliability for programs that execute on unreliable hardware. In: Hosking, A.L., Eugster, P.T., Lopes, C.V. (eds.) OOPSLA. pp. 33–52. ACM (2013)
12. Cook, B., Podelski, A., Rybalchenko, A.: Termination proofs for systems code. In: Schwartzbach, M.I., Ball, T. (eds.) PLDI. pp. 415–426. ACM (2006)
13. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Graham, R.M., Harrison, M.A., Sethi, R. (eds.) POPL. ACM (1977)
14. Gopalakrishnan, G., Haran, A., Lahiri, S., Rakamaric, Z.: Automated differential program verification for approximate computing, unpublished
15. Graf, S., Saidi, H.: Construction of abstract state graphs with PVS. In: Grumberg, O. (ed.) CAV, LNCS, vol. 1254, pp. 72–83. Springer Berlin Heidelberg (1997)
16. Han, J., Orshansky, M.: Approximate computing: An emerging paradigm for energy-efficient design. In: 18th IEEE European Test Symposium. pp. 1–6. IEEE Computer Society (2013)
17. Henzinger, T.A., Jhala, R., Majumdar, R., McMillan, K.L.: Abstractions from proofs. In: Jones, N.D., Leroy, X. (eds.) POPL. pp. 232–244. ACM (2004)
18. Hoare, C.A.R.: Procedures and parameters: An axiomatic approach. In: Engeler, E. (ed.) Symposium on Semantics of Algorithmic Languages. pp. 102–116. Springer Berlin Heidelberg, Berlin, Heidelberg (1971)
19. Kahng, A.B., Kang, S.: Accuracy-configurable adder for approximate arithmetic designs. In: DAC. pp. 820–825. ACM (2012)
20. Krzysztof R. Apt, Frank S. de Boer, E.R.O.: Verification of Sequential and Concurrent Programs. Springer, London (2009)
21. Kugler, L.: Is "good enough" computing good enough? Commun. ACM 58(5), 12–14 (2015)
22. Lahiri, S.K., Rakamarić, Z.: Towards automated differential program verification for approximate computing. In: 2015 Workshop on Approximate Computing Across the Stack (WAX) (2015), <http://sampa.cs.washington.edu/wax2015/papers/lahiri.pdf>
23. Manna, Z., Pnueli, A.: Temporal verification of reactive systems: Progress (1996), draft
24. Misailovic, S., Carbin, M., Achour, S., Qi, Z., Rinard, M.C.: Chisel: reliability- and accuracy-aware optimization of approximate computational kernels. In: Black, A.P., Millstein, T.D. (eds.) OOPSLA. pp. 309–328. ACM (2014)
25. Pauck, F.: Generierung von Eigenschaftsprüfern in einem Hardware/Software-Co-Verifikationsverfahren. Bachelorthesis, Paderborn University (2014)
26. Podelski, A., Rybalchenko, A.: Transition invariants. In: LICS. pp. 32–41. IEEE Computer Society (2004)

27. Sampson, A., Dietl, W., Fortuna, E., Gnanapragasam, D., Ceze, L., Grossman, D.: EnerJ: approximate data types for safe and general low-power computation. In: Hall, M.W., Padua, D.A. (eds.) PLDI. pp. 164–174. ACM (2011)
28. Sery, O., Fedyukovich, G., Sharygina, N.: Interpolation-based function summaries in bounded model checking. In: HVC. pp. 160–175 (2011)
29. Shafique, M., Ahmad, W., Hafiz, R., Henkel, J.: A low latency generic accuracy configurable adder. In: DAC. pp. 86:1–86:6. ACM (2015)
30. Verma, A.K., Brisk, P., Ienne, P.: Variable latency speculative addition: A new paradigm for arithmetic circuit design. In: Proceedings of the conference on Design, automation and test in Europe. pp. 1250–1255. ACM (2008)
31. Wolf, C.: Yosys open synthesis suite. <http://www.clifford.at/yosys/>
32. Ye, R., Wang, T., Yuan, F., Kumar, R., Xu, Q.: On reconfiguration-oriented approximate adder design and its application. In: CAD. pp. 48–54. IEEE Press (2013)
33. Zhu, N., Goh, W.L., Yeo, K.S.: An enhanced low-power high-speed adder for error-tolerant application. In: International Symposium on Integrated Circuits. pp. 69–72. IEEE (2009)